Speculative Threading: Creating New Methods of Thread-Level Parallelization

Antonio González
Director
Intel Barcelona Research Center
Intel Corporation

Table of Contents

(Click on page number to jump to sections)

Speculative Threading: Creating New Methods of Thread-Level Parallelization

Antonio González
Director
Intel Barcelona Research Center
Intel Corporation

Overview: Emerging Usage Models

A new era of computing has begun and the word "multi" provides a key clue. This new era will be characterized by *multi*-core processors enabling single-die *multi*processing of *multi*threaded applications. Signaling the start of this new era is the emergence of new workloads and usage models into mainstream computing. These workloads and new usages will put tremendous demands on future computing platforms at a time when processor performance is somewhat constrained by energy consumption and power dissipation.

Intel is conducting extensive research on some of these new workloads, particularly those associated with recognition, mining and synthesis (RMS). These workloads involve *multi*modal recognition and synthesis over massive data sets, and are representative of many future applications that will require enormous amounts of computing power.

Multi-core processors are essential in helping attain this computing power. They will enable us to capitalize on the thread-level parallelism necessary for achieving increased performance and power efficiency. Unfortunately, current tools and techniques aren't really designed to maximize the parallelism in most programs at a time when software is needed to jump-start this technological leap. Consequently, Intel researchers have been investigating a number of compiler techniques and microarchitectures that can provide highly threaded code through the use of speculative threading.

This article discusses the factors that are driving the need for speculative threading. It will then explain a new speculative threading model developed within Intel's research labs. Preliminary performance results using this model indicate the potential for significant processing gains with only a small increase in power requirements.

The Need for Speculative Threading

We are on the verge of perhaps the most significant transformation of the microprocessor since its creation. Intel's announcement in early 2005 of a microprocessor with two full processing cores ushered in this new era of computing—the era of multi-core. Intel dual-core processors (two execution units packaged in a single processor) are the first step in a massive transition to multi-core computing. Intel is already conducting research on architectures that could hold dozens or even hundreds of processors on a single die.

For almost four decades, Intel has fulfilled the predictions of Moore's Law through innovations in silicon technology. Continuous increases in transistor count will allow Intel to introduce new capabilities in our processors and keep delivering higher performance in every new processor generation. Performance projections based on past growth point toward reaching processing speeds of 1 trillion instructions per second by 2010. To get there, the new era of computing must do some thorough rethinking of basic foundations such as process technology, architecture and software.

Design Challenges and the Move to Multi-Core

One of today's major challenges is maximizing performance in a power and thermal envelope. The increasing activity density, together with leaking currents from ever-more microscopic transistors, is a challenge for the entire industry. This is particularly true when it comes to mobile, battery-operated devices. Multi-core and multithreaded platforms will play a big role in driving power and thermal efficiency. Their ability to exploit thread-level parallelism makes them normally more power-efficient than counterparts exploiting instruction-level parallelism.

Additional challenges will come from variation—the difference in characteristics of "identical" transistors side by side on a die. Quantum mechanical effects that were largely ignored in previous process technology generations have increasing influence at 45 nanometers (nm), 32nm, and smaller. This calls for new circuits and microarchitectures that can adapt to these variations.

Intel is using the transition to multi-core microarchitecture to address these challenges and to develop new optimized platforms for the enterprise, desktop and mobility market segments. This "balanced platform" approach focuses on enhancing all elements in concert so that the integrated features and capabilities contribute to everything from power efficiency to the ideal user experience.

A Timely Solution: Multi-Core and Thread-Level Parallelism

Thread-level parallelism—the ability to simultaneously process multiple instruction streams or threads—can dramatically improve overall performance. Each of these threads can correspond to a different part of a program and runs on one of the multiple hardware contexts available through multi-core and multithreaded designs. For Intel, these designs greatly expand the playing field for performance gains. They enable performance to be increased through:

- Continued advances in materials, basic transistor fabrication, circuit design and core microarchitecture.
- Expanding the number of execution units in processors and improving individual unit architecture.
- Developing new parallelizing approaches.

Handling RMS Workloads

High-end applications tend to trickle down to wider use as mainstream systems develop the performance characteristics required to run them. This will be the case for recognition, mining and synthesis—applications now performed primarily on supercomputers. As powerful multi-core platforms take over the workplace and home, RMS will play a crucial role in enabling these computers to understand and "see" data the way humans do.

Traditionally we have treated "R," "M," and "S" components as independent application classes. For example, computer vision (recognition), data mining (mining) and 3D photorealistic graphics (synthesis) are traditionally considered independent, stand-alone applications. However, an integration of these component applications, if achieved in real-time in an iRMS (interactive RMS) loop, could lead to exciting new usages in everything from the sciences to medicine to business to entertainment. Intel's vision for multi-core processor architectures will include providing optimized hardware for various classes of computation, such as data mining, advanced image processing, speech recognition, natural language processing, and communication protocol processing.

All these workloads will also require significant software innovation. To truly take advantage of multi-core architectures will require that tasks be highly parallelized and split into subtasks that can be processed concurrently on multiple execution units. Consequently, we are aggressively expanding our software activities and working with others in the industry and academia to ensure that developers have the tools, libraries and other infrastructure to take full advantage of our evolving processors. This includes finding ways to parallelize software into hundreds, or in some cases, thousands of threads.

The Promise of Speculative Threading

Today we rely on the software developer to express parallelism in the application, or we depend on automatic tools (compilers) to extract this parallelism. These methods are only partially successful. To run RMS workloads and make effective use of many cores, we need applications that are highly parallel almost everywhere. This requires a more radical approach.

Compilers are a good example. Compilers must provide code that works in all cases. Consequently, any time a compiler has to parallelize two pieces of code, it has to consider all potential dependences. It has to analyze whether one piece of code might write something in a given memory location that another piece of code may read. Unfortunately, in most cases, a compiler doing this only has an approximate view of the memory locations that are being touched by every single instruction. As a result, whenever the compiler has to detect potential dependences, it tends to be over-conservative. If the compiler cannot prove that two instructions are independent, it presumes they are dependent. This means when the compiler generates the code, it assumes a huge number of dependences that don't exist or very rarely exist. The code ends up overly serialized and misses many opportunities for parallelization.

Let's abandon this type of thinking for a minute and consider how we might instead design multi-core platforms that take an application and parallelize it in a "speculative" way. Speculative threads do not necessarily commit and thus, are not necessarily correct. At the time they are spawned, it's not known whether they will work well or not. At some point, they will have to be checked. They might speed up the application or they might be incorrect and just discarded. The key is that there will be a check of their correctness at some point. If they are not correct, no damage will have been done because errant threads are simply squashed and all their activity discarded. There's a built-in safety net so that the processor ends up in the same state it would have been in if the threads had never executed.

Speculative threads could revolutionize how we parallelize applications. Compilers would no longer have to be conservative. Instead, they could be optimistic. Instead of generating code for the worst case, compilers could generate for the common case. The result would be a much higher degree of parallelism and a significant gain in performance.

Speculative Threading in Memory Dependences

One application of speculation would be in memory dependences. As discussed above, compilers are severely limited because they cannot do an exact analysis of which memory locations are going to be touched by every single instruction. Using speculative threads, it could be assumed in many cases that dependences do not exist. Each speculative thread created would then be checked at runtime. If, at some point in time, it is detected that a speculative thread reads a memory location that is later written by a thread that comes earlier in the sequence, that memory violation would be spotted and the speculative thread squashed. In most cases though, the speculative threads would be successful, speeding up the program.

Speculative Threading in Values

Another speculation that has enormous possibilities is values. For this discussion, refer to **Figure 1a**. The rectangle represents a dynamic sequence of instructions. In a conventional sequential processor, such instructions are executed one after another (or sometimes out of order in a small window and then reassembled in order). Suppose for a multi-core platform we want to parallelize the code that is in red with the code that is in yellow so they execute simultaneously. Doing a dependence analysis, today's approaches would find a dependence through variable *A*. Because the yellow part reads from variable *A*, a synchronization would be inserted for the yellow part to wait for the red part to produce this value. The same happens for variable *R1*. In other words, between these two sections there are two true dependences through *R1* and *A*. Consequently, a conventional compiler will insert a synchronization between each pair of potentially dependent instructions.

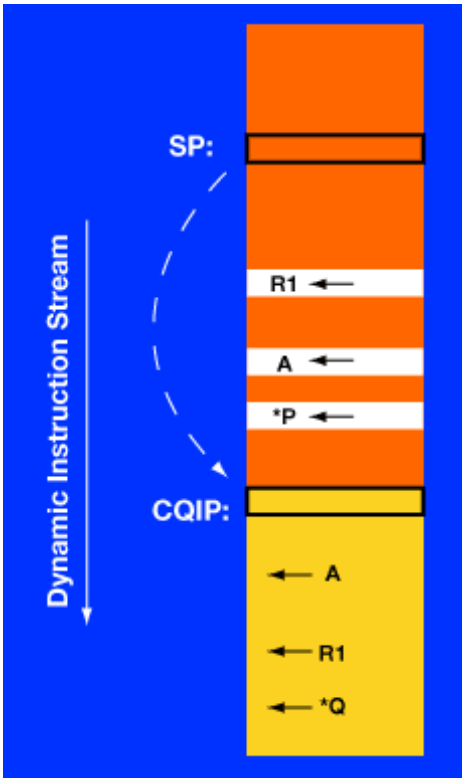

Figure 1a. *This diagram shows the creation of a spawning pair. A spawning pair is defined as a set of two points in the program (each at the beginning of a basic block). The former is called the* spawning point *(SP), marked by the spawn instruction, and it identifies* when *a new speculative thread is created. The latter is called the* control quasi-independent point *(CQIP). It represents* where *the speculative thread starts executing (after some initialization) and is identified as an operand of the spawn instruction.*

But what if there were a way to make a well-reasoned "guess" as to which values these variables are going to take in the red part. Then the processor wouldn't need to wait for these values to be produced and could execute the yellow part in parallel with the red part. When the red part finishes, a check could verify if the guess for these values were correct. If the guesses are correct, everything is fine. The code is parallelized. If the guesses are wrong, the thread is squashed and the yellow part is executed after the red and there's no gain (nor much of anything lost).

The way we're proposing to guess these values is through a software technique called a precomputation slice. The compiler will include a spawning instruction in the red code that will spawn a speculative thread. The speculative thread will start precomputing the values it will most likely need. This precomputation will be done very rapidly by making aggressive (unsafe) code optimizations, since it is not required to be correct. The objective is to generate a slice that computes the thread input values correctly most of the time, but with a much smaller overhead than the one required for a safe computation.

When the red thread arrives at the beginning of the yellow one, the precomputed values are validated, and in case of misspeculation the yellow thread is squashed (see **Figure 1b**).

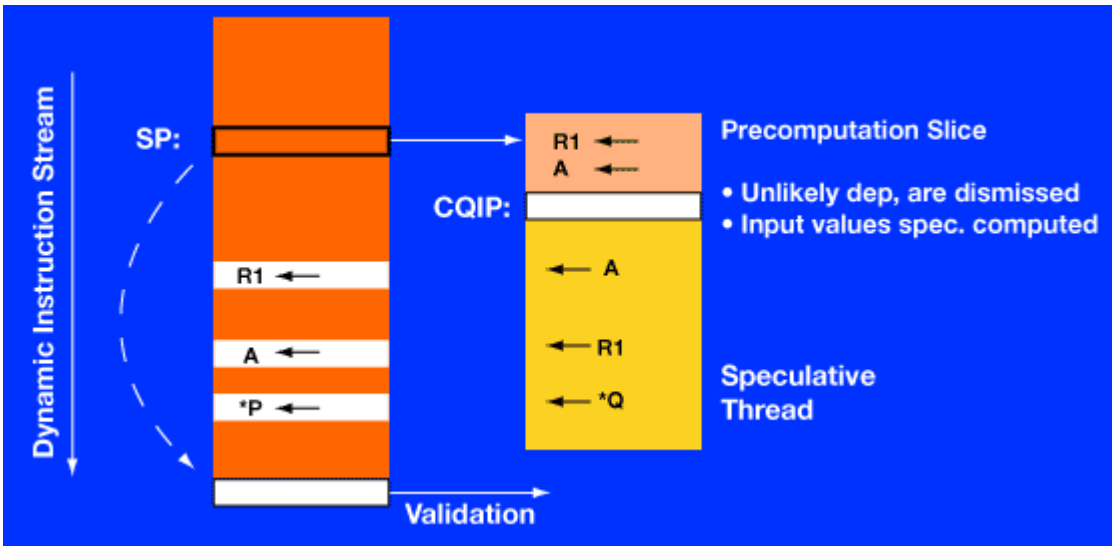

Figure 1b: *Precomputation slices speculatively compute input values.*

This figure also illustrates a speculation on a memory dependence. In this case, there is an ambiguous dependence through pointers *P* and *Q*. If the compiler suspects that this dependence does not occur, it can simply ignore it. At runtime, the hardware will check whether this dependence actually occurs, and if it does, the speculative thread is squashed.

Speculative threads enable the compiler to try numerous types of unsafe, aggressive, optimistic optimizations when generating the precomputation slice. In the end, you'll end up with highly optimized code that can precompute these values very quickly and still be correct most of the time. This is illustrated in **Figure 2**.

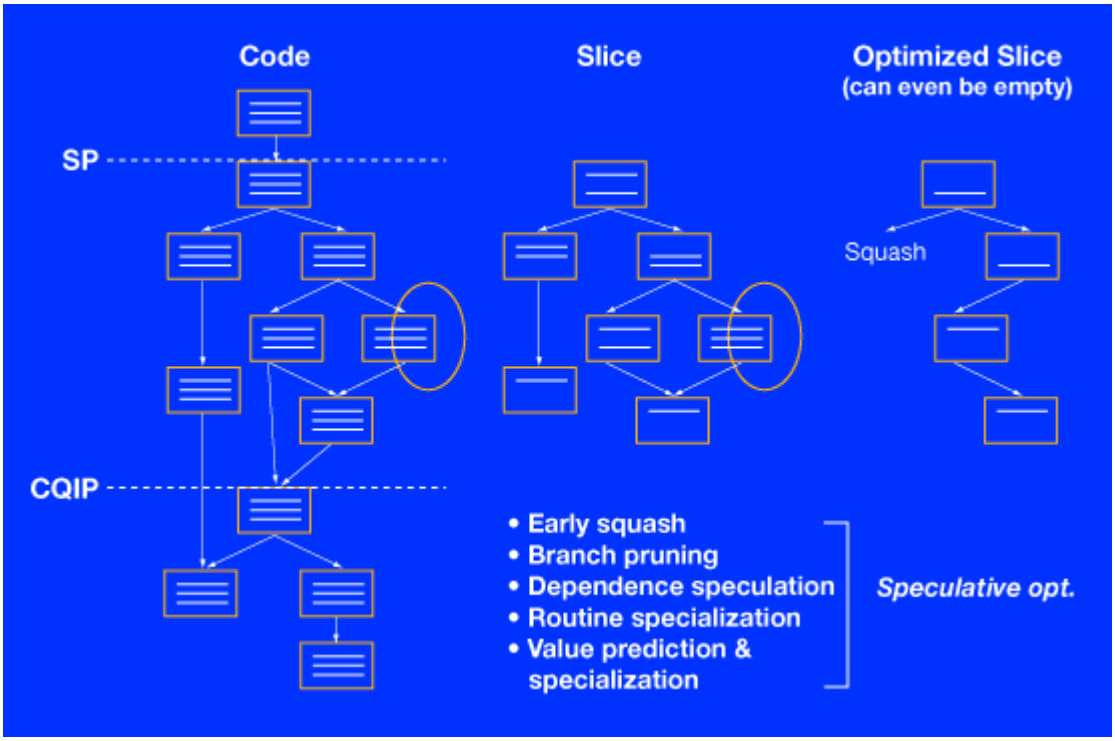

Figure 2. *Through a series of "unsafe" optimizations, our compiler generates a very small slice that still computes the thread input values correctly most of the time at a much faster speed.*

The "Mitosis" System

We've just described how Intel's speculative threading system "Mitosis" works. This codename was chosen for the natural analogy it makes between cells splitting apart and working in parallel and speculative threading.

Mitosis relies on both hardware and software (compiler) support to work. On the software side, the Mitosis compiler is responsible for analyzing the program, and locating the sections of it that can efficiently be executed in parallel. A key component of this analysis is the identification of sections of code whose corresponding precomputation slices have a very low computation overhead. Other conventional aspects such as workload balance also need to be considered.

On the hardware side, Mitosis is built on top of a multi-core and/or multithreaded processor. The main extension required is support for buffering and multiversioning in the memory hierarchy. Buffering is needed to keep the speculative state until the thread is verified and can be committed. Multiversioning is required to allow each variable to have a different value for each of the threads that are running in parallel. This is needed because every thread is executing a piece of code that started out with sequential semantics, but now, parallelized in threads, is being worked on simultaneously with values that were previously supplied in different points in time in the program.

This means the variables of the concurrent threads are the same, but the values these variables contain may be different since they represent the state of the program at different points of the sequential semantics. Mitosis also relies on hardware support to check which variables are read by each of the threads and which memory locations are written. This enables data dependence misspeculations to be spotted quickly, and the corresponding threads be discarded.

The Mitosis system has been designed to optimize the trade-off between software and hardware to exploit speculative thread-level parallelism.

The Results

To illustrate the performance potential of the Mitosis compiler, let's look at a subset of the Olden benchmark suite. Olden benchmarks are pointer-intensive programs that make it difficult for automatic parallel compilers to extract any thread-level parallelism.

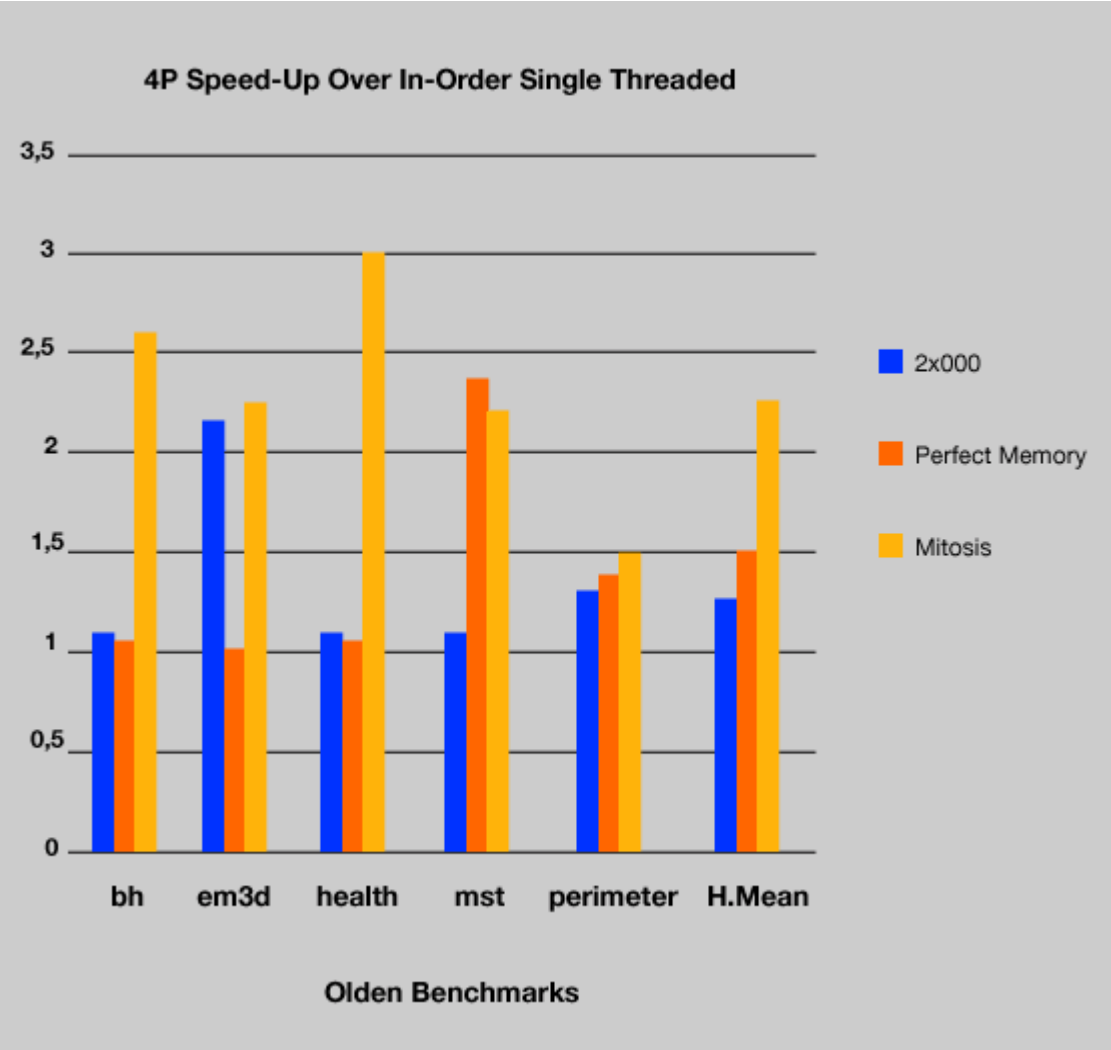

Figure 3. *In this graph, the blue bars show the performance improvement going from an in-order to an out-of-order core with about twice the amount of resources. The red bars indicate the performance of a processor with perfect memory, illustrating the potential performance improvement for any technique that targets simply reducing memory latency. The yellow bars show the performance gains that result when using Mitosis with a four-core processor.*

The results obtained by the Mitosis compiler/architecture for this subset of the Olden benchmarks are very encouraging, outperforming single-threaded execution by 2.2x. When compared with a big out-of-order core, the speed increase is close to 2x. One can also see that the benefits of Mitosis do not come only from reducing memory latency—using Mitosis enables the system to outperform an ideal system with perfect memory by about 60 percent. Overall, this work demonstrates that significant amounts of thread-level parallelism can be exploited in irregular codes, with a rather low overhead in terms of extra (wasted) activity.

Summary

Tomorrow's computing workloads from RMS and other applications of the future will require exponential increases in computing power. Multi-core platforms and multithreading are paving the way to meeting these performance needs. To do it though, they require a companion effort in accelerating the development of the thread-level parallelism necessary for reaching their full performance potential.

The Mitosis technology is a combined software/hardware approach showing great promise in this area. The Mitosis technology's distinguishing feature is the support for speculative threads that the compiler can exploit to perform aggressive optimizations to parallelize code, even if they are unsafe (speculative). In the end, what makes them safe is hardware that provides a safety net to recover correct states whenever needed. The Mitosis technique shows excellent potential for providing additional thread-level parallelism with small power overhead in applications that are hard to parallelize by conventional approaches.

More Info

You can learn much more about Intel microprocessor research at the Intel Web site:

Microprocessor Technology Labs
Intel Barcelona Research Center
Architectural Innovation
Exploratory Research

Read the paper, *Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices*, Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González and Dean M. Tullsen

Author Bio

Antonio González is the director of the Intel Barcelona Research Center, whose research focuses on new microarchitecture paradigms and code generation techniques for future microprocessors. He joined the faculty of the Computer Architecture Department of UPC in 1986 and became a full professor in 2002. His research has focused on computer architecture, compilers and parallel processing, with a special emphasis on processor microarchitecture and code generation. González currently holds a part-time professor position at this department. He has published over 200 papers, has given over 80 invited talks, and has filed 14 patents. González is an associate editor of the IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, ACM Transactions on Architecture and Code Optimization, and Journal of Embedded Computing. He received his M.S. and Ph.D. degrees from the Universitat Politècnica de Catalunya (UPC), in Barcelona, Spain. Read his full bio on the Intel Web site.

—End of Technology@Intel Magazine Article—